#3

P2860X/RSH
06502.0062-01

UNITED STATES PATENT APPLICATION

OF

Peter C. JONES

Ann M. WOLLRATH

James H. WALDO

AND

Kenneth C. R. C. ARNOLD

FOR

DEFERRED RECONSTRUCTION OF OBJECTS AND REMOTE LOADING FOR EVENT NOTIFICATION IN A DISTRIBUTED SYSTEM

REFERENCE TO RELATED APPLICATIONS

The present application is a continuation-in-part of U.S. patent application serial no. 08/950,756, filed on October 15, 1997, and entitled "Deferred Reconstruction of Objects and Remote Loading in a Distributed System," which is incorporated herein by reference.

The following identified U.S. patent applications are relied upon and are incorporated by reference in this application as if fully set forth.

Provisional U.S. Patent Application No. _____, entitled "Distributed Computing System," filed on February 26, 1998.

U.S. Patent Application No. _____, entitled "Method and System for Leasing Storage," bearing attorney docket no. 06502.0011-01000, and filed on the same date herewith.

U.S. Patent Application No. _____, entitled "Method, Apparatus, and Product for Leasing of Delegation Certificates in a Distributed System," bearing attorney docket no. 06502.0011-02000, and filed on the same date herewith.

U.S. Patent Application No. _____, entitled "Method, Apparatus and Product for Leasing of Group Membership in a Distributed System," bearing attorney docket no. 06502.0011-03000, and filed on the same date herewith.

U.S. Patent Application No. _____, entitled "Leasing for Failure Detection," bearing attorney docket no. 06502.0011-04000, and filed on the same date herewith.

U.S. Patent Application No. _____, entitled "Method for Transporting Behavior in Event Based System," bearing attorney docket no. 06502.0054-00000, and filed on the same date herewith.

LAW OFFICES
VNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N.W.
ASHINGTON, DC 20005
202-408-4000

U.S. Patent Application No. _____, entitled "Methods and Apparatus for Remote Method Invocation," bearing attorney docket no. 06502.0102-00000, and filed on the same date herewith.

U.S. Patent Application No. _____, entitled "Method and System for Deterministic Hashes to Identify Remote Methods," bearing attorney docket no. 06502.0103-00000, and filed on the same date herewith.

U.S. Patent Application No. _____, entitled "Method and Apparatus for Determining Status of Remote Objects in a Distributed System," bearing attorney docket no. 06502.0104-00000, and filed on the same date herewith.

U.S. Patent Application No. _____, entitled "Downloadable Smart Proxies for Performing Processing Associated with a Remote Procedure Call in a Distributed System," bearing attorney docket no. 06502.0105-00000, and filed on the same date herewith.

U.S. Patent Application No. _____, entitled "Suspension and Continuation of Remote Methods," bearing attorney docket no. 06502.0106-00000, and filed on the same date herewith.

U.S. Patent Application No. _____, entitled "Method and System for Multi-Entry and Multi-Template Matching in a Database," bearing attorney docket no. 06502.0107-00000, and filed on the same date herewith.

U.S. Patent Application No. _____, entitled "Method and System for In-Place Modifications in a Database," bearing attorney docket no. 06502.0108, and filed on the same date herewith.

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N.W.
WASHINGTON, DC 20005
202-408-4000

U.S. Patent Application No. _____, entitled "Method and System for Typesafe Attribute Matching in a Database," bearing attorney docket no. 06502.0109-00000, and filed on the same date herewith.

Sub C3

U.S. Patent Application No. _____, entitled "Dynamic Lookup Service in a Distributed System," bearing attorney docket no. 06502.0110-00000, and filed on the same date herewith.

U.S. Patent Application No. _____, entitled "Apparatus and Method for Providing Downloadable Code for Use in Communicating with a Device in a Distributed System," bearing attorney docket no. 06502.0112-00000, and filed on the same date herewith.

U.S. Patent Application No. _____, entitled "Method and System for Facilitating Access to a Lookup Service," bearing attorney docket no. 06502.0113-00000, and filed on the same date herewith.

U.S. Patent Application No. _____, entitled "Apparatus and Method for Dynamically Verifying Information in a Distributed System," bearing attorney docket no. 06502.0114-00000, and filed on the same date herewith.

U.S. Patent Application No. 09/030,840, entitled "Method and Apparatus for Dynamic Distributed Computing Over a Network," and filed on February 26, 1998.

Sub C3

U.S. Patent Application No. _____, entitled "An Interactive Design Tool for Persistent Shared Memory Spaces," bearing attorney docket no. 06502.0116-00000, and filed on the same date herewith.

U.S. Patent Application No. _____, entitled "Polymorphic Token-Based Control," bearing attorney docket no. 06502.0117-00000, and filed on the same date herewith.

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N.W.
WASHINGTON, DC 20005
202-408-4000

U.S. Patent Application No. _____, entitled "Stack-Based Access Control," bearing attorney docket no. 06502.0118-00000, and filed on the same date herewith.

Sub C4

U.S. Patent Application No. _____, entitled "Stack-Based Security Requirements," bearing attorney docket no. 06502.0119-00000, and filed on the same date herewith.

U.S. Patent Application No. _____, entitled "Per-Method Designation of Security Requirements," bearing attorney docket no. 06502.0120-00000, and filed on the same date herewith.

FIELD OF THE INVENTION

The present invention relates to a system and method for transmitting objects between machines in a distributed system and more particularly relates to deferred reconstruction of objects for event notification in a distributed system.

BACKGROUND OF THE INVENTION

Distributed programs which concentrate on point-to-point data transmission can often be adequately and efficiently handled using special-purpose protocols for remote terminal access and file transfer. Such protocols are tailored specifically to the one program and do not provide a foundation on which to build a variety of distributed programs (e.g., distributed operating systems, electronic mail systems, computer conferencing systems, etc.).

While conventional transport services can be used as the basis for building distributed programs, these services exhibit many organizational problems, such as the use of different data types in different machines, lack of facilities for synchronization, and no provision for a simple programming paradigm.

LAW OFFICES
NNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
ASHINGTON, DC 20005
202-408-4000

Distributed systems usually contain a number of different types of machines interconnected by communications networks. Each machine has its own internal data types, its own address alignment rules, and its own operating system. This heterogeneity causes problems when building distributed systems. As a result, program developers must include in programs developed for such heterogeneous distributed systems the capability of dealing with ensuring that information is handled and interpreted consistently on different machines.

However, one simplification is afforded by noting that a large proportion of programs use a request and response interaction between processes where the initiator (i.e., program initiating a communication) is blocked waiting until the response is returned and is thus idle during this time. This can be modeled by a procedure call mechanism between processes. One such mechanism is referred to as the remote procedure call (RPC).

RPC is a mechanism for providing synchronized communication between two processes (e.g., program, applet, etc.) running on the same machine or different machines. In a simple case, one process, e.g., a client program, sends a message to another process, e.g., a server program. In this case, it is not necessary for the processes to be synchronized either when the message is sent or received. It is possible for the client program to transmit the message and then begin a new activity, or for the server program's environment to buffer the incoming message until the server program is ready to process a new message.

RPC, however, imposes constraints on synchronism because it closely models the local procedure call, which requires passing parameters in one direction, blocking the calling process (i.e., the client program) until the called procedure of the server program is complete, and then

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

returning a response. RPC thus involves two message transfers, and the synchronization of the two processes for the duration of the call.

The RPC mechanism is usually implemented in two processing parts using the local procedure call paradigm, one part being on the client side and the other part being on the server side. Both of these parts will be described below with reference to FIG. 1.

FIG. 1 is a diagram illustrating the flow of call information using an RPC mechanism. As shown in FIG. 1, a client program 100 issues a call (step 102). The RPC mechanism 101 then packs the call as arguments of a call packet (step 103), which the RPC mechanism 101 then transmits to a server program 109 (step 104). The call packet also contains information to identify the client program 100 that first sent the call. After the call packet is transmitted (step 104), the RPC mechanism 101 enters a wait state during which it waits for a response from the server program 109.

The RPC mechanism 108 for the server program 109 (which may be the same RPC mechanism as the RPC mechanism 101 when the server program 109 is on the same platform as the client program 100) receives the call packet (step 110), unpacks the arguments of the call from the call packet (step 111), identifies, using the call information, the server program 109 to which the call was addressed, and provides the call arguments to the server program 109.

The server program receives the call (step 112), processes the call by invoking the appropriate procedure (step 115), and returns a response to the RPC mechanism 108 (step 116). The RPC mechanism 108 then packs the response in a response packet (step 114) and transmits it to the client program 100 (step 113).

Receiving the response packet (step 107) triggers the RPC mechanism 101 to exit the wait state and unpack the response from the response packet (step 106). RPC 101 then provides the response to the client program 100 in response to the call (step 105). This is the process flow of the typical RPC mechanism modeled after the local procedure call paradigm. Since the RPC mechanism uses the local procedure call paradigm, the client program 100 is blocked at the call until a response is received. Thus, the client program 100 does not continue with its own processing after sending the call; rather, it waits for a response from the server program 109.

The Java™ programming language is an object-oriented programming language that is typically compiled into a platform-independent format, using a bytecode instruction set, which can be executed on any platform supporting the Java virtual machine (JVM). This language is described, for example, in a text entitled "The Java Language Specification" by James Gosling, Bill Joy, and Guy Steele, Addison-Wesley, 1996, which is incorporated herein by reference. The JVM is described, for example, in a text entitled "The Java Virtual Machine Specification," by Tim Lindholm and Frank Yellin, Addison Wesley, 1996, which is incorporated herein by reference. Java and Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Because the JVM may be implemented on any type of platform, implementing distributed programs using the JVM significantly reduces the difficulties associated with developing programs for heterogenous distributed systems. Moreover, the JVM uses a Java remote method invocation (RMI) system that enables communication among programs of the system. RMI is explained in, for example, the following document, which is incorporated herein by reference: Remote Method Invocation Specification, Sun Microsystems, Inc. (1997), which is available via

LAW OFFICES
NNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N.W.
ASHINGTON, DC 20005
202-408-4000

universal resource locator (URL) http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html.

FIG. 2 is a diagram illustrating the flow of objects in an object-oriented distributed system 200 including machines 201 and 202 for transmitting and receiving method invocations using the JVM. In system 200, machine 201 uses RMI 205 for responding to a call for object 203 by converting the object into a byte stream 207 including an identification of the type of object transmitted and data constituting the object. While machine 201 is responding to the call for object 203, a process running on the same or another machine in system 200 may continue operation without waiting for a response to its request.

Machine 202 receives the byte stream 207. Using RMI 206, machine 202 automatically converts it into the corresponding object 204, which is a copy of object 203 and which makes the object available for use by a program executing on machine 202. Machine 202 may also transmit the object to another machine by first converting the object into a byte stream and then sending it to the third machine, which also automatically converts the byte stream into the corresponding object.

The automatic reconstruction of the objects from the byte stream in this manner sometimes requires unnecessary processing. For example, there are times when a call is made that does not require actual or immediate interaction with the object, both of which require conversion of the byte stream to object form. Instead, a call may require passing the object to another call or storing it for later use. In this situation, the reconstruction of the object on an intermediate machine is unnecessary, especially if the object is to be transmitted to another machine. Examples of such a situation include transmission of objects for notification of events

in a distributed system. Accordingly, it is desirable to more efficiently transmit objects in a distributed system without the unneeded conversion of a byte stream to an object on intermediate machines that have no use for the object, or the premature conversion of the byte stream before a process on the receiving machine requires access to the object.

SUMMARY OF THE INVENTION

A method consistent with the present invention specifies an object associated with a request for notification of a particular event within a distributed system. The object is converted into a stream containing a self-describing form of the object, and the stream is provided for selective transmission to a machine where the object is reconstructed by accessing program code identified in the stream upon occurrence of the event.

Another method consistent with the present invention receives at a first machine a stream containing a self-describing form of an object associated with a request for notification of a particular event within a distributed system. The method includes determining whether to send the stream to a second machine and selectively sending the stream to the second machine for reconstruction of the object by accessing program code identified in the stream, the first machine providing notification of the event.

An apparatus consistent with the present invention specifies an object associated with a request for notification of a particular event within a distributed system. The apparatus converts the object into a stream containing a self-describing form of the object and provides the stream for selective transmission to a machine where the object is reconstructed by accessing program code identified in the stream upon occurrence of the event.

Another apparatus consistent with the present invention receives at a first machine a stream containing a self-describing form of an object associated with a request for notification of a particular event within a distributed system. The apparatus determines whether to send the stream to a second machine and selectively sends the stream to the second machine for reconstruction of the object by accessing program code identified in the stream, the first machine providing notification of the event.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings are incorporated in and constitute a part of this specification and, together with the description, explain the advantages and principles of the invention. In the drawings,

FIG. 1 is a diagram illustrating the flow of call information using an RPC mechanism;

FIG. 2 is a diagram illustrating the transmission of objects in an object-oriented distributed system;

FIG. 3 is a diagram of an exemplary distributed processing system that can be used in an implementation consistent with the present invention;

FIG. 4 is a diagram of an exemplary distributed system infrastructure;

FIG. 5 is a diagram of a computer in a distributed system infrastructure shown in FIG. 4;

FIG. 6 is a diagram of a flow of objects within a distributed processing system consistent with the present invention;

FIG. 7 is a flow diagram of steps performed in transmitting objects in a distributed system using loading of remote code for construction of an object in an implementation consistent with the present invention;

LAW OFFICES
INEGAN, HENDERSON,
ARABOW, GARRETT,
& DUNNER, L.L.P.
300 I STREET, N. W.
SHINGTON, DC 20005
202-408-4000

FIG. 8 is a flow diagram of steps performed for deferring code loading and construction of objects when transmitting objects in a distributed system consistent with the present invention;

FIG. 9 is a diagram of a distributed network illustrating event notification; and

FIG. 10 is a flow chart of a process for event notification within a distributed network.

DETAILED DESCRIPTION

Overview

Systems consistent with the present invention efficiently transfer objects using a variant of an RPC or RMI, passing arguments and return values from one process to another process each of which may be on different machines. In such cases, it is desirable to defer reconstruction of the object and downloading of code associated with such object reconstruction until it is needed by the program. The term "machine" is used in this context to refer to a physical machine or a virtual machine. Multiple virtual machines may exist on the same physical machine. Examples of RPC systems include distributed computed environment (DCE) RPC and Microsoft distributed common object model (DCOM) RPC.

An example of how this is accomplished is by making a self-describing stream a first-class entity in the system, meaning that it exists within a type system of a programming language and can be accessed and manipulated by instructions written in that language. A stream is typically a sequence of characters, such as a bit pattern, capable of transmission, and a self-describing byte stream is a byte stream that contains enough information such that it can be converted back into the corresponding object.

An object called a "marshalled object" comprises the self-describing stream. Such marshalled objects can typically be produced from any object that can be passed from one

address space to another, and they can be stored, passed to other objects, or used to reconstruct an object of the original type on demand. The advantage of using marshalled objects is that the reconstruction of an object is deferred until a process having access to the marshalled object directly invokes the creation of the object using the marshalled object. Any downloading of code required to operate on the object is deferred until the marshalled object is used to create a copy of the original object, which was previously used to produce the marshalled object.

Accordingly, in cases where the object is not used, but rather is stored for later retrieval or passed along to another process, RMI does not download the code required for reconstruction of the object. This may result in considerable efficiencies, both in time and in code storage space.

Event notification, for example, may occur through use of a marshalled object. For event notification, a machine registers with a device to receive notification of particular events within a distributed network. The device transmits a request for registration along with a marshalled object to an event generator, which stores the marshalled object for possible later transmission. If the event occurs, the event generator sends notification of the event including the marshalled object to an event listener. The event listener may reconstruct the marshalled object, which may contain information relating to the event. The event listener may be the same as the device requesting notification. Events include, for example, a change in the state or occurrence of an object. More specific examples of events in a distributed system include, but are not limited to, the following: a "click" by a key or cursor-control device; an overlapping window on a display device; a device joining a network; a user logging onto a network; and particular user actions.

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

Distributed Processing System

FIG. 3 illustrates an exemplary distributed processing system 300 which can be used in an implementation consistent with the present invention. In FIG. 3, distributed processing system 300 contains three independent and heterogeneous platforms 301, 302, and 303 connected in a network configuration represented by network cloud 319. The composition and protocol of the network configuration represented by cloud 319 is not important as long as it allows for communication of the information between platforms 301, 302 and 303. In addition, the use of just three platforms is merely for illustration and does not limit an implementation consistent with the present invention to the use of a particular number of platforms. Further, the specific network architecture is not crucial to embodiments consistent with this invention. For example, another network architecture that could be used in an implementation consistent with this invention would employ one platform as a network controller to which all the other platforms would be connected.

In the implementation of distributed processing system 300, platforms 301, 302 and 303 each include a processor 316, 317, and 318 respectively, and a memory, 304, 305, and 306, respectively. Included within each memory 304, 305, and 306, are applications 307, 308, and 309, respectively, operating systems 310, 311, and 312, respectively, and RMI components 313, 314, and 315, respectively.

Applications 307, 308, and 309 can be applications or programs that are either previously written and modified to work with, or that are specially written to take advantage of, the services offered by an implementation consistent with the present invention. Applications 307, 308, and

LAW OFFICES
NNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
ASHINGTON, DC 20005
202-408-4000

309 invoke operations to be performed in accordance with an implementation consistent with this invention.

Operating systems 310, 311, and 312 are typically standard operating systems tied to the corresponding processors 316, 317, and 318, respectively. The platforms 301, 302, and 303 can be heterogenous. For example, platform 301 has an UltraSparc® microprocessor manufactured by Sun Microsystems, Inc. as processor 316 and uses a Solaris® operating system 310. Platform 302 has a MIPS microprocessor manufactured by Silicon Graphics Corp. as processor 317 and uses a Unix operating system 311. Finally, platform 303 has a Pentium microprocessor manufactured by Intel Corp. as processor 318 and uses a Microsoft Windows 95 operating system 312. An implementation consistent with the present invention is not so limited and could accommodate homogenous platforms as well.

Sun, Sun Microsystems, Solaris, Java, and the Sun Logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. UltraSparc and all other SPARC trademarks are used under license and are trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

Memories 304, 305, and 306 serve several functions, such as general storage for the associated platform. Another function is to store applications 307, 308, and 309, RMI components 313, 314, and 315, and operating systems 310, 311, and 312 during execution by the respective processor 316, 317, and 318. In addition, portions of memories 304, 305, and 306 may constitute shared memory available to all of the platforms 301, 302, and 303 in network

319. Note that RMI components 313, 314, and 315 operate in conjunction with a JVM, which is not shown for the purpose of simplifying the figure.

Distributed System Infrastructure

Systems and methods consistent with the present invention may also operate within a particular distributed system 400, which will be described with reference to FIGS. 4 and 5. This distributed system 400 is comprised of various components, including hardware and software, to (1) allow users of the system to share services and resources over a network of many devices; (2) provide programmers with tools and programming patterns that allow development of robust, secured distributed systems; and (3) simplify the task of administering the distributed system. To accomplish these goals, distributed system 400 utilizes the Java programming environment to allow both code and data to be moved from device to device in a seamless manner. Accordingly, distributed system 400 is layered on top of the Java programming environment and exploits the characteristics of this environment, including the security offered by it and the strong typing provided by it.

In distributed system 400 of FIGS. 4 and 5, different computers and devices are federated into what appears to the user to be a single system. By appearing as a single system, distributed system 400 provides the simplicity of access and the power of sharing that can be provided by a single system without giving up the flexibility and personalized response of a personal computer or workstation. Distributed system 400 may contain thousands of devices operated by users who are geographically disperse, but who agree on basic notions of trust, administration, and policy.

Within an exemplary distributed system are various logical groupings of services provided by one or more devices, and each such logical grouping is known as a Djinn. A

"service" refers to a resource, data, or functionality that can be accessed by a user, program, device, or another service and that can be computational, storage related, communication related, or related to providing access to another user. Examples of services provided as part of a Djinn include devices, such as printers, displays, and disks; software, such as programs or utilities; information, such as databases and files; and users of the system.

Both users and devices may join a Djinn. When joining a Djinn, the user or device adds zero or more services to the Djinn and may access, subject to security constraints, any one of the services it contains. Thus, devices and users federate into a Djinn to share access to its services. The services of the Djinn appear programmatically as objects of the Java programming environment, which may include other objects, software components written in different programming languages, or hardware devices. A service has an interface defining the operations that can be requested of that service, and the type of the service determines the interfaces that make up that service.

Distributed system 400 is comprised of computer 402, a computer 404, and a device 406 interconnected by a network 408. Device 406 may be any of a number of devices, such as a printer, fax machine, storage device, computer, or other devices. Network 408 may be a local area network, wide area network, or the Internet. Although only two computers and one device are depicted as comprising distributed system 400, one skilled in the art will appreciate that distributed system 400 may include additional computers or devices.

FIG. 5 depicts computer 402 in greater detail to show a number of the software components of distributed system 400. One skilled in the art will appreciate that computer 404 or device 406 may be similarly configured. Computer 402 includes a memory 502, a secondary

storage device 504, a central processing unit (CPU) 506, an input device 508, and a video display 510. Memory 502 includes a lookup service 512, a discovery server 514, and a Java runtime system 516. The Java runtime system 516 includes the Java RMI system 518 and a JVM 520. Secondary storage device 504 includes a Java space 522.

As mentioned above, distributed system 400 is based on the Java programming environment and thus makes use of the Java runtime system 516. The Java runtime system 516 includes the Java API libraries, allowing programs running on top of the Java runtime system to access, in a platform-independent manner, various system functions, including windowing capabilities and networking capabilities of the host operating system. Since the Java API libraries provides a single common API across all operating systems to which the Java runtime system is ported, the programs running on top of a Java runtime system run in a platform-independent manner, regardless of the operating system or hardware configuration of the host platform. The Java runtime system 516 is provided as part of the Java software development kit available from Sun Microsystems, Inc. of Mountain View, CA.

JVM 520 also facilitates platform independence. JVM 520 acts like an abstract computing machine, receiving instructions from programs in the form of bytecodes and interpreting these bytecodes by dynamically converting them into a form for execution, such as object code, and executing them. RMI 518 facilitates remote method invocation by allowing objects executing on one computer or device to invoke methods of an object on another computer or device. Both RMI and the JVM are also provided as part of the Java software development kit.

Lookup service 512 defines the services that are available for a particular Djinn. That is, there may be more than one Djinn and, consequently, more than one lookup service within distributed system 400. Lookup service 512 contains one object for each service within the Djinn, and each object contains various methods that facilitate access to the corresponding service. Lookup service 512 is described in U.S. patent application entitled "Method and System for Facilitating Access to a Lookup Service," which was previously incorporated herein by reference.

Discovery server 514 detects when a new device is added to distributed system 400, during a process known as boot and join (or discovery), and when such a new device is detected, the discovery server passes a reference to lookup service 512 to the new device so that the new device may register its services with the lookup service and become a member of the Djinn. After registration, the new device becomes a member of the Djinn, and as a result, it may access all the services contained in lookup service 512. The process of boot and join is described in U.S. patent application entitled "Apparatus and Method for providing Downloadable Code for Use in Communicating with a Device in a Distributed System," which was previously incorporated herein by reference.

A Java space 522 is an object repository used by programs within distributed system 400 to store objects. Programs use a Java space 522 to store objects persistently as well as to make them accessible to other devices within distributed system 400. Java spaces are described in U.S. patent application serial no. 08/971,529, entitled "Database System Employing Polymorphic Entry and Entry Matching," assigned to a common assignee, and filed on November 17, 1997, which is incorporated herein by reference. One skilled in the art will appreciate that an

LAW OFFICES
NEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
SHINGTON, DC 20005
202-408-4000

exemplary distributed system 400 may contain many lookup services, discovery servers, and Java spaces.

Data Flow in a Distributed Processing System

FIG. 6 is a diagram of an object-oriented distributed system 600 connecting machines 601, 602, and 603, such as computers or virtual machines executing on one or more computers, or the machines described with reference to FIG. 3. Transmitting machine 601 includes a memory 604 storing objects such as objects 605 and 606, and RMI 607 for performing processing on the objects. To transmit an object over network 600, RMI 607 uses code 609 for converting object 605 into a marshalled object that is transmitted as a byte stream 608 to machine 602. Streams used in the Java programming language, including input and output streams, are known in the art and an explanation, which is incorporated herein by reference, appears in, for example, a text entitled "The Java Tutorial: Object-Oriented Programming for the Internet," pp. 325-53, by Mary Campione and Kathy Walrath, Addison-Wesley, 1996.

Part of this conversion includes adding information so that a receiving machine 602 can reconstruct the object. When a set of object types is limited and is the same on all machines 601, 602, and 603, a receiving machine typically requires the object's state and a description of its type because the object's code is already present on all network machines. Alternatively, machine 601 uses RMI 607 to provide more flexibility, allowing code to be moved when necessary along with information or the object's state and type. Additionally, a transmitting machine includes in the marshalled object an identification of the type of object transmitted, the data constituting the state of the object, and a network-accessible location in the form of a URL for code that is associated with the object. URLs are known in the art and an explanation, which

is incorporated herein by reference, appears in, for example, a text entitled "The Java Tutorial: Object-Oriented Programming for the Internet," pp. 494-507, by Mary Campione and Kathy Walrath, Addison-Wesley, 1996.

When receiving machine 602 receives byte stream 608, it identifies the type of transmitted object. Machine 602 contains its own RMI 610 and code 611 for processing of objects. If byte stream 608 contains a marshalled object, machine 602 may create a new object 614 using the object type identified in the marshalled object, the state information, and code for the object. Object 614 is a copy of object 605 and is stored in memory 613 of machine 602. If code 612 is not resident or available on machine 602 and the marshalled object does not contain the code, RMI 610 uses the URL from the marshalled object to locate the code and transfer a copy of the code to machine 602. Code 612 is generally only required if the object is unmarshalled. Because the code is in bytecode format and is therefore portable, the receiving machine can load the code into RMI 610 to reconstruct the object. Thus, machine 602 can reconstruct an object of the appropriate type even if that kind of object has not been present on the machine before.

Machine 602 may also convert object 614 into byte stream 615 for transmission to a third machine 603, which contains its own RMI 618 and code 619 for processing objects. RMI 618, using code 620 for the object, converts byte stream 615 into a corresponding object 616, which it stores in memory 617. Object 616 is a copy of object 605. If code 620 for the object is not resident or available, machine 603 requests the code from another machine using the URL, as described above.

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

Machine 602 may alternatively store the marshalled object as a byte stream without reconstructing the object. It may then transmit the byte stream to machine 603.

Marshalled Object

A marshalled object is a container for an object that allows that object to be passed as a parameter in RMI call, but postpones conversion of the marshalled object at the receiving machine until a program executing on the receiving machine explicitly requests the object via a call to the marshalled object. A container is an envelope that includes the data and either the code or a reference to the code for the object, and that holds the object for transmission. The serializable object contained in the marshalled object is typically serialized and deserialized when requested with the same semantics as parameters passed in RMI calls. Serialization is a process of converting an in-memory representation of an object into a corresponding self-describing byte stream. Deserialization is a process of converting a self-describing byte stream into the corresponding object.

To convert an object into a marshalled object, the object is placed inside a marshalled object container and when a URL is used to locate code for the object, the URL is added to the container. Thus, when the contained object is retrieved from its marshalled object container, if the code for the object is not available locally, the URL added to the container is used to locate and load code in bytecode format for the object's class.

Table 1 provides an exemplary class definition in the Java programming language for a marshalled object consistent with the present invention.

Table 1

```
package java.rmi;
public final class MarshalledObject implements java.io.Serializable
{
        public MarshalledObject (Object obj)
                throws java.io.IOException;

        public Object get ()
                throws java.io.IOExeception, ClassNotFoundException;

        public int hashCode ();

        public boolean equals();
}
```

A marshalled object may be embodied within an article of manufacture specifying a representation of the object stored in a computer-readable storage medium.

A marshalled object's constructor takes a serializable object (obj) as its single argument and holds the marshalled representation of the object in a byte stream. The marshalled representation of the object preserves the semantics of objects that are passed in RMI calls: each class in the stream is typically annotated with either the object's code or a URL to the code so that when the object is reconstructed by a call to a "get" method, the bytecodes for each class can be located and loaded, and remote objects are replaced with their proxy stubs. The "get" method is a method called by a program to execute a process of unmarshalling, which is reconstruction of an object from a marshalled object using a self-describing byte stream (the marshalled object), and a process to obtain the necessary code for that process. A proxy stub is a reference to a remote object for use in reconstructing an object.

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

When an instance of the class marshalled object is written to a "java.io.ObjectOutputStream," the contained object's marshalled form created during construction is written to the stream. Thus, only the byte stream is serialized.

When a marshalled object is read from a "java.io.ObjectInputStream," the contained object is not deserialized into a new object. Rather, the object remains in its marshalled representation until the marshalled object's get method is called.

The "get" method preferably always reconstructs a new copy of the contained object from its marshalled form. The internal representation is deserialized with the same semantics used for unmarshalling parameters for RMI calls. Thus, the deserialization of the object's representation loads class codes, if not available locally, using the URL annotation embedded in the serialized stream for the object.

As indicated in the class definition for a marshalled object, the hash code of the marshalled representation of an object is defined to be equivalent to the hash code for the object itself. In general, a hash code is used in hash tables to perform fast look-ups of information, which is known in the art. The equals method will return true if the marshalled representation of the objects being compared are equivalent. An equals method verifies reconstruction by determining if a reconstructed object is the same as the original object, and such methods are known in the Java programming language.

Transmission of a Marshalled Object

FIG. 7 is a flow diagram of steps 700 preferably performed in transmitting objects in a distributed system consistent with the present invention. A machine receives a byte stream (step 701), which includes data for the object, information identifying the type of object, and

optionally a URL for the code that is associated with the object. The receiving machine determines if the code for the object is resident or available (step 702). If it is available, the machine preferably uses RMI for reconstructing the object from the byte stream and resident code (step 704). If the code is not resident, the machine uses the URL from the byte stream to request the code from another machine located at a network accessible location, and that machine returns a copy of the code (step 703). The object can also be transmitted in the form of a byte stream to another machine (step 705).

FIG. 8 is a flow diagram of steps 800 preferably performed for deferring code loading and construction of objects when transmitting marshalled objects in a distributed system consistent with the present invention. A machine receives a byte stream (step 801), which includes data for the object, information identifying the type of object, and optionally a URL for the code that is associated with the object.

The machine determines if the byte stream is a marshalled object (step 802). If it is not such an object, the machine performs normal processing of the byte stream (step 803). Otherwise, if the received byte stream represents a marshalled object, the machine holds the marshalled object for later use in response to a get method invoked by a process on the receiving machine. If the receiving machine determines that the object is to be transmitted to another machine (step 804), it simply transmits the byte stream without reconstructing the object. If the machine uses the object, it performs reconstruction of the object using its RMI and associated code (step 805). If the reconstruction code for the object is not resident on the machine, it uses a URL to request and obtain the code (step 806), as described above. The machine determines if it

needs to transmit the object to another machine (step 807). If the object is destined for another machine, it is transmitted as a byte stream (step 808).

Accordingly, a marshalled object provides for more efficient transfer of objects in a distributed system. If an object is needed by a machine, it can be reconstructed, and if the machine does not need to use the object, it can transmit the marshalled object without reconstructing it.

Use of a Marshalled Object in Event Notification

A distributed system or network may use marshalled objects in conjunction with registration for notification of events within the system. FIG. 9 is a diagram of a distributed network 900 illustrating event notification. Network 900 may use the machines described with reference to FIGS. 3, 4, and 5. Network 900 includes a remote event listener 901 having RMI 902 and object 903, a machine 904 having RMI 905 and object 906, and an event generator 907 having RMI 908 and object 909 for providing event notification. Machine 904 may be the same as remote event listener 901, as indicated by the dashed line, or they may be separate machines.

Machine 904, desiring notification of a particular network event, registers with RMI 908 by transmitting a request for event notification including or associated with a marshalled object 912. Event generator 907 stores the marshalled object for possible later transmission. When RMI 908 detects an occurrence of the event, it transmits notification of the event along with the marshalled object 913 to remote event listener 901. Remote event listener 901 may make a call to code server 910 in order to obtain code 911 for reconstructing the marshalled object, which may contain information relating to the event. A code server is an entity and process that has access to code and responds to requests for a particular type or class of object and returns code

for that object. A code server may be located within machine 901 or on another machine. Also, the code may be resident on the same platform as the code server or on a separate platform. After registration, remote event listener 901 may provide indications of the particular event when detected.

FIG. 10 is a flow chart of a process 1000 or event notification within a distributed network, such as network 900. A machine sends a registration request including a marshalled object to a remote event generator (step 1001), which stores the marshalled object for possible later transmission (step 1002). The event generator determines if the event occurred (step 1003), and, if it detects such an occurrence, it sends notification of the event including the stored marshalled object to an event listener machine (step 1004). The event generator determines, based on particular criteria or system requirements, if it is to continue providing notification of occurrences of the event (step 1005). If so, it continues to determine if the event occurred. These steps may use, for example, interfaces and class definitions written in the Java programming language shown in Tables 1-5 provided in this specification.

Table 2 provides an example of an interface in the Java programming language for implementing a remote event listener.

Table 2

```
public interface RemoteEventListener extends Remote,
                                        java.util.EventListener
{
    void notify(RemoteEvent theEvent)
        throws EventUnknownException, RemoteException;
}
```

In the interface shown in Table 2, the notify method has a single parameter of type "RemoteEvent" that is used during operation to encapsulate the information passed as part of a notification. Table 3 provides an example of a definition for the public part of the "RemoteEvent" class definition.

Table 3

```
public class RemoteEvent extends EventObject
{
        public RemoteEvent (Object evSource,
                            long  evIdNo,
                            long  evSeqNo)
        public Object getSource ( );
        public long getID( );
        public long getSeqNo ( );
        public MarshalledObject getRegistrationObject ( );
}
```

In the definition shown in Table 3, the abstract state contained in a "RemoteEvent" object includes a reference to the object in which the event occurred, a "long" which identifies the kind of event relative to the object in which the event occurred, and a "long" which indicates the sequence number ("SeqNo") of this instance of the event kind. The sequence number obtained from the "RemoteEvent" object is an increasing value that may provide an indication concerning the number of occurrences of this event relative to an earlier sequence number.

Table 4 provides an example of an interface in the Java programming language for an event generator.

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

Table 4

```
public interface EventGenerator extends Remote
{
        public EventRegistration register (long evId,
                                           MarshalledObject handback,
                                           RemoteEventListener toInform,
                                           long leasePeriod)
            throws EventUnknownException, RemoteException;
}
```

The register method shown in Table 4 allows registration of interest in the occurrence of an event inside an object. While executing this method, a JVM receives an "evId," which is a long integer used to identify the class of events; an object that is transmitted back as part of the notification; a reference to a "RemoteEventListener" object; and a long integer indicating the leasing period for the interest registration. If an "evId" is provided to this call that is not recognized by the event generator object, the JVM signals an error. In the Java programming language, a JVM is said to "throw" an "EventUnknownException" to signal that error. The second argument of the register method is a marshalled object that is to be transmitted back as part of the notification generated when an event of the appropriate type occurs. The third argument of the register method is a remote event listener object that receives any notifications of instances of the event kind occurring. This argument may be the object that is registering interest, or it may be another remote event listener such as a third-party event handler or notification "mailbox." The final argument to the register method is a "long" indicating the requested duration of the registration, referred to as the "lease."

The return value of the register method is an object of the event registration class definition. This object contains a "long" identifying the kind of event in which interest was

LAW OFFICES
FINNEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
1300 I STREET, N. W.
WASHINGTON, DC 20005
202-408-4000

registered relative to the object granting the registration, a reference to the object granting the registration, and a lease object containing information concerning the lease period.

Table 5 provides an example of a class definition in the Java programming language for event registration.

Table 5

```
public class EventRegistration implements java.io.Serializable {
public EventRegistration (long eventNum,
        Remote registerWith,
        Lease eventLease,
        long currentSeqNum);
        public long getEventID ( );
public Object getEventSource ( );
public Lease getLease ( );
public long currentSeqNum ( );
}
```

In the class definition shown in Table 5, the "getEventID" method returns the identifier of the event in which interest was registered, which combined with the return value of the "getEventSource" method uniquely identifies the kind of event. This information is provided to third-party repositories to allow them to recognize the event and route it correctly. During operation in a JVM, the "getLease" method returns the lease object for this registration and is used in lease maintenance. The "currentSeqNo" method returns the value of the sequence number on the event kind that was current when the registration was granted, allowing comparison with the sequence number in any subsequent notifications. A "toString" method may be used with this class definition to return a human-readable string containing the information making up the state of the object.

Machines implementing the steps shown in FIGS. 7, 8, and 10 may include computer processors for performing the functions, as shown in FIGS. 3, 4, and 5. They may include modules or programs configured to cause the processors to perform the above functions. They may also include computer program products stored in a memory. The computer program products may include a computer-readable medium or media having computer-readable code embodied therein for causing the machines to perform functions described above. The media may include a computer data signal embodied in a carrier wave and representing sequences of instructions which, when executed by a processor, cause the processor to securely address a peripheral device at an absolute address by performing the method described in this specification. The media may also include data structures for use in performing the method described in this specification.

Although the illustrative embodiments of the systems consistent with the present invention are described with reference to a computer system implementing the Java programming language on the JVM specification, the invention is equally applicable to other computer systems processing code from different programming languages. Specifically, the invention may be implemented with both object-oriented and nonobject-oriented programming systems. In addition, although an embodiment consistent with the present invention has been described as operating in the Java programming environment, one skilled in the art will appreciate that the present invention can be used in other programming environments as well.

While the present invention has been described in connection with an exemplary embodiment, it will be understood that many modifications will be readily apparent to those skilled in the art, and this application is intended to cover any adaptations or variations thereof.

LAW OFFICES
FINEGAN, HENDERSON,
FARABOW, GARRETT,
& DUNNER, L.L.P.
300 I STREET, N.W.
WASHINGTON, DC 20005
202-408-4000

For example, different labels or definitions for the marshalled object may be used without departing from the scope of the invention. This invention should be limited only by the claims and equivalents thereof.